Examen conv. 1 curso 98-99 **2 Febrero 1999** **90 minutos**

Apellidos y Nombre: .....

Problema 1: Búsquedas no informadas *[20 min.]*

Se muestra abajo el código utilizado en la práctica 3 para expandir un nodo. Se pide detallar todas las modificaciones necesarias en la función `expandir-nodo` para que dicha función utilice una nueva función denominada `expandir-estado` que recibe un estado y devuelve la lista de nuevos estados (sin verificar todavía que se encuentren en alguna de las listas de nodos). Los operadores se encontrarán en la lista `*operadores*`, que será la que recorra `expandir-estado` para encontrar los nuevos estados. No volvais a escribir trozos de código que ya se encuentren en el listado, podéis indicarlo con alguna referencia.[1]

```
(defun expandir-nodo (nodo)
  (let* ((estado (get nodo 'estado))
         )
    (format t "~%--->~S" estado)
    (mapcan #'(lambda (x)
                (let ((nuevo-estado (mueve x estado)))
                  (and nuevo-estado
                       (not (member nuevo-estado nodos-expandidos
                                    :test #'equal
                                    :key #'(lambda (y)
                                             (get y 'estado))))
                       (not (member nuevo-estado nodos-a-expandir
                                    :test #'equal
                                    :key #'(lambda (y)
                                             (get y 'estado))))
                       (progn (format t "~%Aniadiendo ~S" nuevo-estado)
                              (list (crea-nodo nuevo-estado nodo))))))
            '((m) (c) (m m) (c c) (m c)))))

 (defun mueve (personas estado)
  (block fuera-del-block
    (let* ((mi (misioneros-izq estado))   ;misioneros en la ribera izquierda
           (ci (canibales-izq estado))    ;canibales en la ribera izquierda
           (bi (bote-izq estado))         ;bote en la ribera izquierda
           (bd (bote-der estado))         ;bote en la ribera derecha
           (cd (canibales-der estado))    ;canibales en la ribera derecha
           (md (misioneros-der estado))  ;misioneros en la ribera derecha
           )

      (cond ((eq bi 'bote)                          ;el bote esta a la izquierda
             (dolist (persona personas)
               (when (eq persona 'm)
                 (if mi (push (pop mi) md) (return-from fuera-del-block nil)))
               (when (eq persona 'c)
                 (if ci (push (pop ci) cd) (return-from fuera-del-block nil))))
             )
            ((eq bd 'bote)                          ;el bote esta a la derecha
             (dolist (persona personas)
               (when (eq persona 'm)
                 (if md (push (pop md) mi) (return-from fuera-del-block nil)))
               (when (eq persona 'c)
                 (if cd (push (pop cd) ci) (return-from fuera-del-block nil))))
             ))
      (cond ((and mi (> (length ci) (length mi))) nil)  ;sobrenumero en la izq
            ((and md (> (length cd) (length md))) nil)  ;sobrenumero en la der
            (t (list 'ribera-izq mi ci bd '--rio-- bi cd md 'ribera-der))))))
```

[1] **NOTA**: No duplicar el código que ya se encuentre escrito en el listado, hacer una adecuada referencia señalizando oportunamente

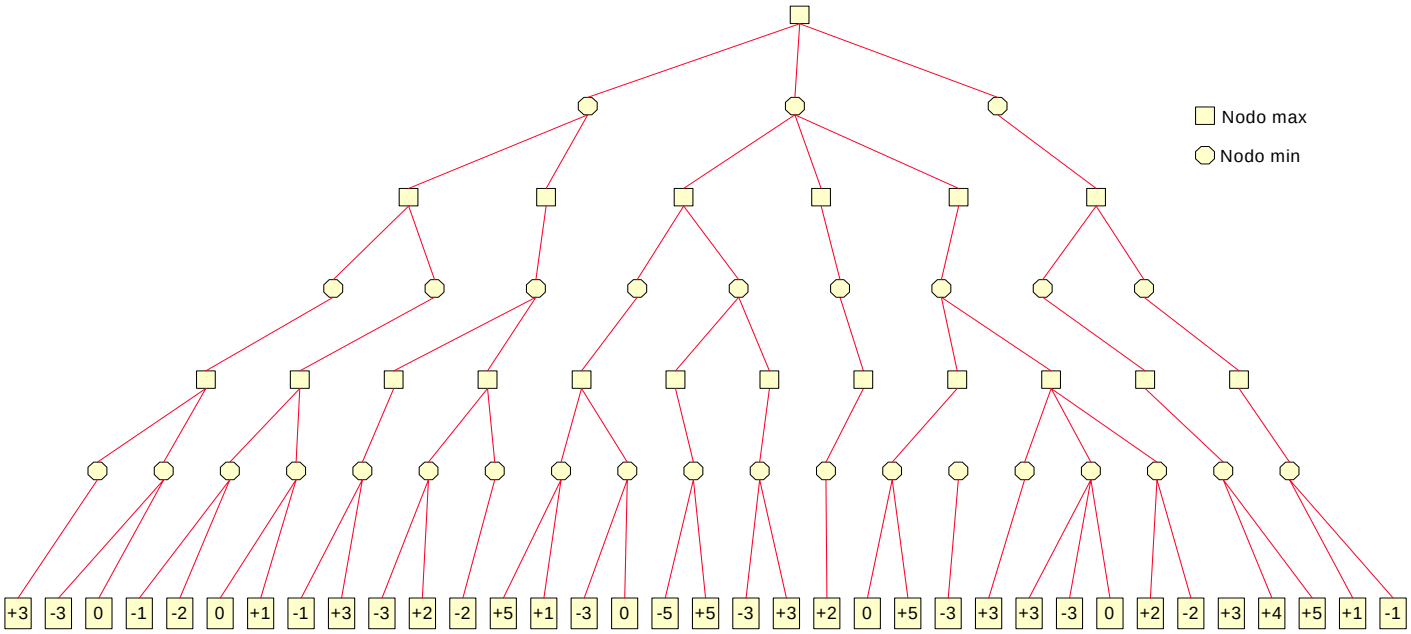
[2]
```
(defun expandir-nodo (nodo)
```

```
(defun expandir-estado (estado)
```

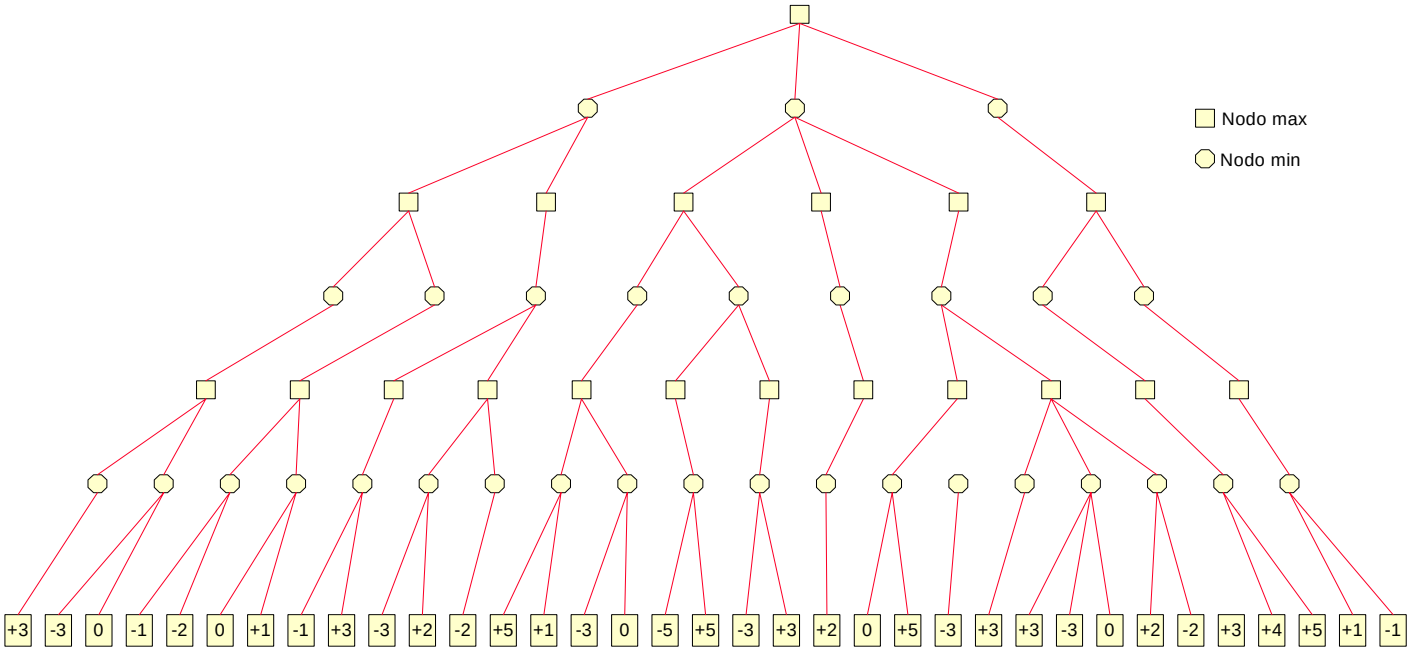
```
(setq *operadores*
```

[2] **NOTA**: No duplicar el código que ya se encuentre escrito en el listado, hacer una adecuada referencia señalizando oportunamente

Problema 2: Búsqueda en juegos *[15 min.]*
Aplica el algoritmo minimax con poda alfa-beta al árbol mostrado en la figura (la búsqueda en profundidad es de izquierda a derecha). Reflejar en el árbol los valores finales de alfa y beta de cada nodo.



Aplícalo ahora con una búsqueda de derecha a izquierda.



NOTA: **Alfa**: (para un nodo MAX) es el valor más alto visto hasta el momento de los valores finales, calculados hacia atrás, de sus sucesores. **Beta**: (para un nodo MIN) es el valor más bajo visto hasta el momento de los valores finales, calculados hacia atrás, de sus sucesores.

Poda (fin de la llamada recursiva): Puede suspenderse la exploración por debajo de cualquier nodo MIN que tenga valores de Beta menores o iguales que el valor Alfa de cualquiera de sus nodos MAX ascendientes suyos. Puede suspenderse la exploración por debajo de cualquier nodo MAX que tenga valores de Alfa mayores o iguales que el valor Beta de cualquiera de sus nodos MIN ascendientes suyos.

Problema 3: Representaciones estructuradas *[20 min.]*
[Tendrá gran valor en la nota del problema que el código resulte claro, sencillo y elegante]

Para la base de conocimiento de la inmobiliaria utilizada en la práctica 5, crea un slot denominado cantidad con el demon `if-needed` adecuado de forma que devuelva la cantidad que en ese momento existan de instancias (forms que no tienen descendientes) de la clase (o instancia) a la que se pregunta[3]. Por ejemplo, para la jerarquía dada de la inmobiliaria:

```
(get-value 'vivienda 'cantidad)
---> 3
```

El código utilizado en la práctica es el siguiente:

```
(defun form (&key name is-a slots)
  (let* ((la-form (if (get name 'is-a) name
                      (progn (setf (get name 'is-a) is-a) name))))
    (dolist (slot-conten slots)
      (funcall #'set-aspect la-form
               (car slot-conten) (cadr slot-conten) (cadr(cdr slot-conten))))
    la-form))

(defun set-aspect (form slot aspecto valor)
  (let* ((cont (get form slot)))
    (cond (cont
           (if (assoc aspecto cont)
             (rplacd (assoc aspecto cont) valor)
             (setq cont (cons (list aspecto valor) cont ))))
          (t
           (setq cont (cons (list aspecto valor) cont ))))
    (setf (get form slot) cont )))

(defun get-aspect (form slot aspecto)
  (let* ((value (assoc aspecto (get form slot))))
    (if value
      (cadr value)
      (if (get form 'is-a)
        (get-aspect (get form 'is-a) slot aspecto)
        ))))

(defun get-value (form slot)
  (let ((needed (get-aspect form slot 'if-needed)))
    (if needed
      (funcall needed form slot)
      (get-aspect form slot '=))))

(defun gbd ()
  (let ()
    (create-form :is-a nil :name 'object)
    (form :name 'cosa :is-a 'object)

    (form :name 'vivienda :is-a 'cosa)

    (form :name 'apartamento :is-a 'vivienda
          :slots (list (list 'ruido-calle 'if-needed #'calcula-ruido)))
    (form :name 'chalet :is-a 'vivienda
          :slots (list (list 'ruido-calle 'if-needed #'calcula-ruido-en-casa)))
    (form :name 'apt-breton-22
          :is-a 'apartamento
          :slots '((calle-nombre = Breton) (calle-numero = 22) (color-pared = blanco)
                   (material-suelo = madera)))
    (form :name 'apt-22-1 :is-a 'apt-breton-22
          :slots '((numero-habitaciones = 3) (piso = 2)))
    (form :name 'chalet-montesol-5 :is-a 'chalet
          :slots '((calle-nombre = urbanizacion-montesol) (calle-numero = 5)
                   (numero-habitaciones = 5) (pisos = 2) (distancia-carretera = 200)))))
(defun calcula-ruido (form slot)
  (let* ((valor (get-aspect form 'ruido-calle '=))
         piso)
    (cond (valor valor)
          (t (setq piso (get-value form 'piso))
             (cond ((> piso 15) 'muy-bajo) ((> piso 8) 'bajo) ((> piso 4) 'medio)
                   ((> piso 1) 'alto) (t 'muy-alto))))))
```

[3] **NOTA**: No duplicar el código que ya se encuentre escrito en el listado, hacer una adecuada referencia señalizando oportunamente

4

[4] **NOTA**: No duplicar el código que ya se encuentre escrito en el listado, hacer una adecuada referencia señalizando oportunamente

Problema 4: Sistemas basados en reglas *[15 min.]*
El siguiente programa CLIPS resuelve el típico problema del granjero, el lobo, la cabra y la col. El programa muestra todas las soluciones encontradas. Lee el programa y completa las reglas que se indican en negrita.

```
(deffunction opuesta (?orilla)
   (if (eq ?orilla izquierda)
      then derecha
      else izquierda))
(deftemplate nodo
   (slot localizacion-granjero )
   (slot localizacion-lobo)
   (slot localizacion-cabra)
   (slot localizacion-col)
   (multislot camino))
(deffacts nodo-inicial
     (nodo
          (localizacion-granjero izquierda)
          (localizacion-lobo izquierda)
          (localizacion-cabra izquierda)
          (localizacion-col izquierda)
          (camino)))

(defrule movimiento-solo
  ?nodo <- (nodo


=>
  (duplicate[5] ?nodo


)
(defrule movimiento-con-lobo



=>



)
(defrule movimiento-con-cabra



=>


)
(defrule movimiento-con-col



=>


)
```

[5] **NOTA**: Duplicate es como modify (incluso la misma sintaxis) pero crea un nuevo nodo.

```
(defrule lobo-come-cabra
  (declare (salience 10000))
  ?nodo <- (nodo (localizacion-granjero ?l1)
                 (localizacion-lobo ?l2&~?l1)
                 (localizacion-cabra ?l2))
  => ...)
(defrule cabra-come-col
  (declare (salience 10000))
  ?nodo <- (nodo (localizacion-granjero ?l1)
                 (localizacion-cabra ?l2&~?l1)
                 (localizacion-col ?l2))
  => ...)
(defrule camino-circular
  (declare (salience 10000))
  (nodo  (localizacion-granjero ?granjero)
         (localizacion-lobo ?lobo)
         (localizacion-cabra ?cabra)
         (localizacion-col ?col)
         (camino $?movimientos-1))
  ?nodo <- (nodo (localizacion-granjero ?granjero)
                 (localizacion-lobo ?lobo)
                 (localizacion-cabra ?cabra)
                 (localizacion-col ?col)
                 (camino $?movimientos-1 ? $?movimientos-2))
  =>
  (retract ?nodo))


(defrule reconoce-solucion
  (declare (salience 10000))
  ?nodo <- (nodo  (localizacion-granjero derecha)
                  (localizacion-lobo derecha)
                  (localizacion-cabra derecha)
                  (localizacion-col derecha)
                  (camino $?movimientos))
  =>
  (retract ?nodo)
  (assert (solucion $?movimientos)))

(defrule escribe-solucion
  ?mv <- (solucion $?m)
  =>
  (retract ?mv)
  (printout t crlf crlf  "Solucion encontrada " crlf)
  (bind ?orilla derecha)
  (loop-for-count (?i 1 (length $?m))
     (bind ?cosa (nth ?i $?m))
     (printout t "El granjero se mueve "
        (switch ?cosa
           (case solo then "solo ")
           (case lobo then "con el lobo")
           (case cabra then "con la cabra ")
           (case col then "con la col "))
          " a la " ?orilla "." crlf)
      (bind ?orilla (opuesta ?orilla))))
```